Autumn Leaves: Curing the Window Plague in IDEs

David Röthlisberger, Oscar Nierstrasz, Stéphane Ducasse

Autumn Leaves: Curing the Window Plague in IDEs

Accepted at WCRE 2009

David Röthlisberger
Software Composition Group
University of Bern, Switzerland

Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland

Stéphane Ducasse
INRIA-Lille Nord Europe
France

***Abstract*—Navigating large software systems is difficult as the various artifacts are distributed in a huge space, while the relationships between different artifacts often remain hidden and obscure. As a consequence, developers using a modern interactive development environment (IDE) are forced to open views on numerous source artifacts to reveal these hidden relationships, leading to a crowded workspace with many opened windows or tabs. Developers often lose the overview in such a cluttered workspace as IDEs provide little support to get rid of unused windows. *AutumnLeaves* automatically selects windows unlikely for future use to be closed or grayed out while important ones are displayed more prominently. This reduces the number of windows opened at a time and adds structure to the developer's workspace. We validate *AutumnLeaves* with a benchmark evaluation using recorded navigation data of various developers to determine the prediction quality of the employed algorithms.**

Keywords: development environments, software navigation, developer activity analysis, productivity, program comprehension

I. Introduction

Object-oriented language characteristics such as inheritance and polymorphism can lead to conceptually related code being scattered over many different source artifacts [4], [22]. This can lead to an unfocused, undirected navigation of the source space. Empirical experiments have shown that during a one day coding session, developers browsed 95% of all visited methods more than once [16]. These navigation difficulties become obvious in an IDE where developers are forced to open many windows or tabs in order to locate entities of interest. IDEs do not show how these windows are related to each other, thus developers are often confronted with an immense number of independent, allegedly unrelated windows or tabs to reason about [3]. It is unclear which windows are still important and which ones have been opened to explore a branch of the navigation space not leading to the final goal. Thus developers are usually uncertain when a window will not be used anymore and are thus not willing to take the risk of closing windows potentially needed in the future [21]. As a result, the workspace becomes cluttered with too many windows.

However, having many windows open at a given moment negatively impacts navigation efficiency as developers have to spend more time locating the window of interest and as they need to keep a larger, more complex mental map of the content and purpose of each open window. Thus it is clearly desirable to have a minimal set of open windows at any point in time, which is likely to reduce time to navigate and maintain the working set of artifacts. However, it is challenging to determine this minimal set, that is, the windows containing relevant, important content useful for the current problem to be solved by the developer.

As navigation is an important prerequisite to program comprehension, improving source space navigation in the IDE is an important step to better understand and reverse engineer applications while they are being developed and maintained [21], [3], [10]. Literature reports that developers spend up to 35% of their time navigating software [11] and up to 60% is spent with program comprehension activities in general [1], [2].

In this paper we propose *AutumnLeaves*, an enhancement for IDEs such as Eclipse or Smalltalk to automatically close windows unlikely to be used in the future. To achieve this goal *AutumnLeaves* determines the likelihood of a window's content to be of use to the developer by relating it to all other opened artifacts. If for instance a window contains a class, a window showing a related class (such as a super- or subclass) or a method of this class is related to the first window. *AutumnLeaves* assigns to every open window a weight that will be increased upon every navigation action in the same or any other window that is related to the content showed in this window. This weight allows *AutumnLeaves* to identify those windows that have no references or only weak ones to the current development task performed by the developer. Windows with relatively little weight are steadily grayed out until *AutumnLeaves* closes them automatically (optionally by asking the developer for confirmation beforehand). This closing action occurs when the weight of a window compared to all other weights drops below a certain threshold. *AutumnLeaves* thus acts as a garbage collector for windows to mitigate the window plague with which developers are typically confronted in modern IDEs.

The research question addressed in this paper is how to model hidden references between the various windows opened in a development session to be able to generally determine the importance of windows and in particular to identify futile, unused windows, similar to the way a garbage collector locates and terminates unreferenced objects. How should we model references between very different windows

used in software development (code, debugger, inspector, or references windows) and how to represent importance of windows and changes in importance during a development task?

This paper addresses these questions by first reporting on the plague of too many opened windows in software development in Section II. Second, we introduce *AutumnLeaves*, our proposal to model window references and to detect obsolete windows in Section III. We validate *AutumnLeaves* in Section IV concerning correctness and practicability by conducting a benchmark validation based on 25 recorded development sessions. We analyze theses sessions to determine whether *AutumnLeaves* correctly closed a window or whether the developer used this window after *AutumnLeaves* would have closed it. Section V discusses differences between common window management techniques employed in IDEs. Section VI reports on related work in the field of easing navigation of the source space in IDEs. Finally, Section VII concludes the paper and reports on future work.

II. Window Plague in IDEs

Most software systems spread their functionality over multiple source artifacts. Even reasonably sized systems contain several hundreds of these artifacts (classes, methods). Depending on the programming language these artifacts are contained in files (for instance in Java or C/C++) or are directly accessible as objects in languages such as Smalltalk [7]. In any case, developers navigating these artifacts in modern IDEs such as Eclipse [5], a Smalltalk IDE [7] or any other environment, usually view and navigate source entities by opening windows or tabs. Normally one window or tab only shows one single source entity at a time.

As soon as a window has been opened to view an artifact, it is unclear whether and how long this view is required to complete the development task. Thus developers are usually reluctant to quickly close windows, instead they keep the views on the artifacts open as they fear to not be able to easily recover these views once closed. As a consequence, they open more and more windows, in particular when working on complex, object-oriented applications whose code is scattered over many different artifacts in statically distinct and disperse parts of the code base (for instance, in multiple packages) [4], [22].

We conducted several small empirical surveys and studies with developers either working with Java in Eclipse or with Smalltalk in Squeak [9]. The fundamental difference between these two IDEs is that Eclipse works on the basis of files containing Java classes while Squeak contains classes and methods as first-class entities not stored in files. Squeak thus supports the direct navigation of methods without first opening the declaring file and class therein. Eclipse also employs the concept of tabs (see Figure 1) while in Squeak, developers open full-fledged windows arranged on a desktop. These windows can be moved, resized and minimized and often serve themselves as full-fledged browsers (that is, they contain the entire package tree from packages down to methods).

Metric	Eclipse	Squeak
Number of windows opened	35.84	25.74
Avg. number of open windows	16.68	14.29
Number of windows closed	10.35	12.96
Number of windows opened and closed shortly thereafter	2.24	4.15
Number of window switches	58.90	38.85
Number of entities revisited	41.64	35.10

Table I
Characteristic of the window plague in the Eclipse and Smalltalk IDE

In our empirical studies we analyzed typical development sessions of developers working on smaller projects (applications with up to 100 classes of either Java or Smalltalk code). We recorded the number of opened windows in total, the average number of open windows (measured in intervals of five minutes), the number of windows closed, and the number of windows opened, browsed and closed just afterwards (without changing focus to another window or tab). Additionally, we recorded the number of times developers switched from one window to another and how often they visited a previously browsed entity again without editing this entity on re-visit, that is, to just read and understand it again. The development sessions recorded lasted for half an hour for each developer. In total we analyzed 22 such development sessions. Table I reports on the findings of these studies.

These numbers highlight the fact that developers usually open many more windows than they close, thus the list of opened windows steadily grows. This results in an average number of open windows higher than human beings are capable to cognitively handle. We can assume that due to this high number of open windows, developers lose the overview and their navigation efficiency is hampered. The high number of window switches or number of entities visited several times are indications for lost overview and confusion resulting from being overloaded with plenty of windows in the development workspace.

As Eclipse employs the concept of tabs and does not use full-fledged browser windows as Squeak, it is in general easier to re-find windows in Eclipse as they do not overlap. However, even in Eclipse the developer usually only sees between five to ten tabs in the tab bar on the screen. To access remaining open windows it is necessary to use the list next to the tab bar (see Figure 1). Developers reported to us that locating a window of interest in this list is very difficult and time-consuming. Usually they opt to not use this window list, but to navigate to the appropriate source artifact in the package tree and open again a view on it; Eclipse then automatically opens the window already displaying this artifact.

Figure 1. Eclipse supports tabbed browsing of the source space, but there is only space for a limited number of tabs; additional tabs are accessible in scroll list on the right.

A common pattern of most interviewed developers to deal with the window plague is to let the list of windows grow until they are completely done with the current task. Then developers take the time to manually close all or most windows opened during the task solving process. Very few developers close windows they consider as not needed anymore regularly during a task. However, such a procedure leads to a constantly growing list of windows clearly hampering navigation efficiency. Developers reported spending a considerable amount of time whenever they have to re-locate an open window. Moreover, they are aware that most windows they have opened become useless over time, but they are not willing or able to manually close the windows most likely not to be needed anymore.

III. AUTUMNLEAVES

AutumnLeaves is an approach to overcome the previously discussed window plague. First we explain the basic principles behind *AutumnLeaves* and we report on several design considerations and variation points. The ultimate goal of *AutumnLeaves* is to identify unused windows, that is, autumn leaves that can fall down from the tree as they are not useful anymore and push away by the wind.

A. AutumnLeaves in a Nutshell

AutumnLeaves associates a weight to each open window to indirectly model references between windows. This weight is increased upon certain user actions. Also the entities displayed in any window have a weight. This is necessary to relate entities with windows. If for instance one window displays a class, another a method or a subclass of this class, we add in our model an implicit reference between these two windows based on the entities they show. We keep the entity weight even if windows containing such entities are closed. This enables us to re-establish references between windows when the developer again opens a view on this entity in a new window.

To identify obsolete, useless windows, the weight of each window is compared to the average weight of all open windows. If a window weight is below a certain threshold of the average weight (defined as 30%), *AutumnLeaves* suggests to close the window. This suggestion is visually displayed by graying out the window or its title bar in case of tabs. Developers can always decline the automatic closing of a window, otherwise the window is closed five user actions after falling below the threshold. Additionally, the current window weight is steadily displayed in the right corner of a window to make developers aware of candidate windows for removal.

The weights (for a complete list see III) and the threshold are determined by performing a benchmark validation on recorded data sets of navigation and modification activities performed by several developers working on various development tasks. Section IV reports in detail about this benchmark validation. In a nutshell, we assume that *AutumnLeaves* performs well if it does close windows not used anymore later in the recorded development session. According to that idea, we ran the benchmark on 25 recorded development sessions and ultimately selected the best performing threshold and weights. We had to trade off correctness (not closing windows used later on) against effectiveness of *AutumnLeaves* (measured with the reduction in average number of open windows) and favored correctness if the results between two weight configurations were similar. We have chosen the initial weight configurations (how much specific actions should increase weight) based on our personal experience for the importance of actions and varied the concrete weight around the initially chosen weight for each action by two weight points up and down.

For the threshold we experimented with all values from 5% to 50% in 5% intervals. We discovered that the effect of *AutumnLeaves*, that is, the reduction of number of windows, drops quickly when lowering the threshold while correctness remains relatively stable. However, when the threshold raises above 30%, the correctness value starts to drop fast, hence we have chosen to close a window when its weight falls below 30% of the average window weight. This threshold could be further optimized, but we consider 30% as a reasonable value.

The weights of all windows are refreshed and checked against the threshold after each user action. As a user action we consider opening a window, typing or scrolling in a window, moving or minimizing windows. To determine entity weights, we additionally consider viewing (“opening”), creating, modifying, and deleting methods and classes. The final weight of a window is the sum of its own weight and the weight of the entity it currently displays. If the displayed entity is a single method, we also add the weight of its class to the window weight (only applicable for Smalltalk as we cannot open views on single methods in Java).

To build references between windows we mostly use the entities displayed in a window. If we modify a method, we increase the weight for this entity, but also for the containing class. We thus propagate weight according to

Action	Class	Method	Propagation
Viewing (“opening”)	3	3	1.5
Modifying	8	10	4
Creating	4	4	2
Removing	-	-	2

Table II
WEIGHT ADDITION TO SOURCE ENTITIES UPON CERTAIN ACTIONS ON THE SAME OR DEPENDENT ENTITIES. PROPAGATION MEANS ADDING WEIGHT TO RELATED ENTITIES, FOR INSTANCE FROM A METHOD TO ITS CLASS OR FROM A CLASS TO ITS SUPERCLASS.

Action	Weight addition
Initial opening	12
Moving	1
Resizing	1
Getting focus	2
Typing in it	8
Visibility (in Squeak also fractions thereof)	1

Table III
WEIGHT ADDITION TO THE A WINDOW UPON CERTAIN ACTIONS ON THIS WINDOW.

static relationships between source artifacts: From a method to its class, from a class to its direct superclass and all direct subclasses, from an inner class to its outer class, from an interface to all implementing classes. Propagated weight is always half of the direct weight for the entity: If we add weight 10 to a method, its class gets 5 points. Table II lists the different weights for all actions on entities, Table III for window actions. With these settings we obtained best results concerning correct identification of unused windows and reduction of number of windows.

Some IDEs allow developers to hide or overlap windows with others. In Squeak for instance, windows can overlap and partially or fully hide windows behind. In Eclipse, only a limited number of tabs is visible on the screen. Older tabs are only visible in the drop-down menu to the right of the tab bar. We consider visible windows to be more important than hidden ones. Thus we reward fully visible windows or tabs with an additional weight point after every user action. In Squeak, we additionally take into account the degree of visibility, that is, the portion of the window being at the front on the desktop and add the visible proportion of one weight point to the window weight on every user action. The desktop management facility of Squeak allows windows to be stacked.

To make sure that the weighting mechanism also properly handles windows in which no navigation actions happens but that are just selected to view their contents, we increase the weight of a window by two points when obtaining the focus. This weight is only given when the developer looks for more than three seconds at the window to only reward windows the developer inadvertently selected.

We consider all kinds of windows dealing with entire source entities, that is, class browsers (showing classes and methods), debuggers, inspectors, workspaces (for code snippets), list windows (list of class references, method senders or implementors, variable references, etc.). The window has to focus on a particular entity, that is, one single class or one single method. In Eclipse we consider the method in the center of the source view as the selected method. For Eclipse views such as the package explorer or the type view we consider just the selected entity but not other visible entities close to the selected one. If the entire list shown in a list view such as the package explorer was considered, we could not easily identify relations between different windows based on displayed source artifacts as most windows would be related to each other when using the entire content of list views. Other types of windows such as simple text editors, file browsers, or XML editors are not handled by *AutumnLeaves* and will thus never be automatically closed.

B. Variation Points

Pinning of windows. One variation point is a pinning facility for windows. A window manually pinned by the developer will never be closed by *AutumnLeaves*. It will always stay there even if its weight has dropped below the threshold. Such a feature is useful for windows serving as libraries or documentation. Developers might never type in these windows, maybe not even interact with them, but still they serve a purpose to show content of interest to developers, content that is permanently important, such as a list of constants. Thus the pinning mechanism makes sure that such reference windows can stay. Developers are free to pin any kind of windows and as many as they want. The pinning mechanism also makes sure that the windows opened for a specific task do not get closed by *AutumnLeaves* when interrupting this task to work on something else. For instance, the pinning could be categorized, so that all windows for the same tasks can be identified by the pinning category.

Visibility of windows. In Squeak, windows can overlap other windows. Thus the visibility of a window, that is, whether it is fully visible at the front, partially visible because of other windows covering it, or totally hidden by other windows, certainly has an influence on the importance of a window to the developer. We can assume that a fully hidden window at the end of the stack is less likely to be used by the developer than a (partially) visible window. Maybe the developer even forgot about the existence of such a hidden window. We currently account for this fact by rewarding visible windows with additional weight points on each user action, in Squeak depending on the extent of visibility. In Eclipse a tab is either fully visible or fully hidden, thus a visible window always obtains a full reward point. However, another mechanism to take into account visibility could be to check visibility just at the moment *AutumnLeaves* actually suggests to close a particular window. We defined two thresholds at which windows should be closed: a higher boundary for hidden or partially hidden windows

(*e.g.*, 40%) and a lower boundary for visible windows (*e.g.*, 20%). We experimented with both mechanisms and report in Section IV-A on differences between these two concerning correctness.

Weighting previously selected entities. Another variation point is how viewed entities should influence the weight of a window. In particular in Squeak, developers often navigate entities directly in particular windows as most windows provide browser facilities to navigate source code (Eclipse differs here as its windows only provide local navigation facilities, for instance scrolling from one method of a class to the next). Thus the importance of such a window not only depends on the currently displayed source artifact, but also on the recent history of therein navigated artifacts. As Squeak offers means to easily navigate the history of a browser window, similar to functionality provided by web browsers, a previously viewed class is still conveniently accessible from within this window. If this displayed class is important and many other windows refer to it, then this particular window should have a higher importance even when the developer navigates further to a particular method of this class, as the old viewed entity is still easily accessible from within this window. We thus take into account in Squeak not just the currently selected entity, but also the two artifacts navigated before this entity when computing the weight of a window. The window weight is thus the sum of the weight of the window itself and the weights of the first three entities in the window navigation history. We have chosen the number three and not more to be able to react to changes in development focus, for instance if open windows are reused for a new exploration path, previous entities should not influence the window weight for too long.

Weights. The weights we have chosen (see Table II and Table III) are another, important variation point. The rational behind the currently defined weights is to particularly take into account the content displayed in windows, that is, the navigated source artifacts, classes and methods, to be able to relate different windows to each other. However, as a variation we can also put more emphasis on actions performed on the windows themselves, such as the time spent in a window (for typing, scrolling, or having the focus). The emphasis on the entities can be further relaxed by not propagating weight from an entity to related entities (for instance, from a method to its declaring class). Section IV-A discusses the impact of weight propagation to related entities.

IV. Validation

In this section we validate our work in two basic directions: First, we perform a benchmark validation to study the correctness of *AutumnLeaves*, that is, whether our approach correctly identifies candidate windows to be closed. Second, we report on the practicality of *AutumnLeaves*, that is, how developers assess its usefulness in practice, when working on concrete tasks in their daily work.

A. Correctness

To evaluate the correct and desired functioning of *AutumnLeaves*, that is, identifying the appropriate candidate windows for closing, we performed a benchmark validation. A benchmark validation has the advantage of being easily replicable, it eases the comparison of results, and can be used to test a restricted functionality, such as the effect of different weights on the performance of *AutumnLeaves*. The same validation procedure has been used by other researchers to evaluate similar works such as code completion engines [18].

Procedure. In a nutshell, the benchmarking procedure we implemented replays a recorded sequence of user interactions occurred in the IDE. After each action, we let *AutumnLeaves* compute the weight of all windows as discussed in Section III. If the algorithms identify a candidate window for removal, we look forward in the recorded user actions whether the developer ever used this window again and if so, what kind of actions he performed in this window.

In total, we analyzed 25 recorded development sessions of eight different developers. Each development session lasted between half an hour and three hours. In these sessions, very different tasks have been performed in different software systems. The development sessions used in this evaluation are not the same as those mentioned in Section II to make the results more generalizable and less tailored to the data used to identify the problem we want to solve with *AutumnLeaves*. The sessions used for the validation are longer and more complex in terms of application and task size than those used in Section II. Also the developers are different persons, except one developer who contributed different recorded sessions to this evaluation as well as to the initial identification of the problem. Most developers are either graduate or PhD students that worked on various tasks in research projects. We asked developers that we personally know to install our recording tool in their IDE and to submit us recorded sessions of any kind. The recording tool we implemented instruments the IDE code to send announcements about all navigation and modification activities occurring in each window we are interested in. In this validation benchmark we iterate over all recorded data sessions to find out for each window when it has been lastly used. In a second iteration we evaluate after each recorded action whether *AutumnLeaves* suggests to close a window and check whether it has still be used by the developer afterwards.

The participating developers described us what kind of tasks they performed in the respective session. From these descriptions we identified six different task categories: Implementing a new system from scratch (2 sessions), implementing a new feature for an existing application with which the developer was either familiar (3) or unfamiliar (4), fixing a defect in a system (7), optimizing a system's performance (1), and a pure navigation task to gain an

initial understanding for an unfamiliar software system (8). A new feature implementing task was for instance to add a navigation history button showing all previously navigated source artifacts in the Squeak browser. One navigation task for example was concerned with determining the classes responsible for rendering arrowed lines between figures in a drawing program. Most of the 25 development sessions stem from development in Squeak, while only a few (three sessions) originate from Eclipse. The systems on which developers were working had a size of approximately hundred up to five hundred classes, except the application that has been developed from scratch. After evaluating the general performance of *AutumnLeaves* we specifically test whether this performance depends on the nature of the task being performed in a development session.

The best result for the performance of *AutumnLeaves* is certainly if the developer never again used the window *AutumnLeaves* suggested to close. Even if he used the window later on in the recorded activity log, we analyze how often the window has been used and whether it has been used to navigate or modify the same or a related entity (for instance, method or subclass of a class, a class in the same package of a class, etc.). If the window was later on used to navigate something completely different, we rate the decision of *AutumnLeaves* as correct as the developer could also have opened an entirely new window instead of re-using an existing one. If the window has been used to work on the same or on related artifacts, we count the related actions performed in this window and give *AutumnLeaves* a correctness rating of the reciprocal value of the counted user actions in this particular window. If *AutumnLeaves* for instance suggests to close a window that has been used ten times afterwards, we give this decision a correctness value of 0.1. If the window has been used just once, the decision is still considered as fully correct. However, if a window has been used more than 10 times, we rate the decision of *AutumnLeaves* as entirely wrong. To obtain the final correctness rate for *AutumnLeaves* in a particular development session, we summed up all correctness rates for all candidate windows *AutumnLeaves* suggested to close and divided this by the number of total candidates.

Results. Table IV shows the correctness results we got for different development sessions. Due to space restrictions we do not show all 25 but just five selected sessions, and the total performance averaged over all 25 sessions. The five selected tasks are in this order: New feature implementation (Squeak), defect correction (Squeak), navigation (Squeak), performance optimization (Eclipse), navigation (Eclipse). This table also shows the correctness value if computed in an "all or nothing" manner: Only considering a window to be closed that is never used anymore afterwards is rated as a correct performance of *AutumnLeaves*. Thus this correctness value is the percentage of perfectly correctly identified windows to be closed. We also analyzed the data sets to identify windows that have not been suggested by *AutumnLeaves* for removal, but have not been used after a certain moment. These windows can be considered as false-negatives as *AutumnLeaves* should have identified them as well. We also determined the average time between the last usage of a window and the moment *AutumnLeaves* was able to pinpoint a window to be closed. This measure gives evidence on how fast *AutumnLeaves* is able to detect changes in the direction the development takes, for instance if the developer explores another, unrelated branch of the source space. A window is considered as being used until the end of the session if the last hundred user actions involved this window; such a window is thus not a false-negative. Furthermore, we give details about the reduction of the average number of open windows (measured in intervals of five minutes).

Discussion of the results. The results in Table IV show that *AutumnLeaves* usually correctly closed windows when a few usages of a window closed by *AutumnLeaves* are still acceptable, that is, these usages reduce the correctness rate just reciprocally to the number of usages. However, the correctness value dropped significantly when only closing a window never used later on is acceptable. Nonetheless, we can still trust the suggestions of *AutumnLeaves* as those windows have not been used often after *AutumnLeaves* suggested their closing and hence cannot possibly have played a crucial role in the development session.

With 6.12 windows, the average number of false negatives is pretty low. We consider this as a very promising performance of *AutumnLeaves*, in particular when comparing with the average number of opened windows (65.20). However, it takes *AutumnLeaves* a considerable amount of time (on average more than 10 minutes) to identify a window not used in the future. This means that with the current weighting mechanism, it is difficult to react on quickly changing directions in development focus. If for instance the developer finished exploring a part of the application (*e.g.*, the database layer), it takes time until this is reflected in the content displayed in the various windows. The developer has to navigate further in most windows or even manually close old windows in order to make *AutumnLeaves* aware of the new development focus. We will tackle this problem in future work.

The reduction of the number of average open windows (minus 12.50) is also a positive sign for the performance of *AutumnLeaves*. We can consider any reduction of the number of open windows to be an improvement, provided that truly obsolete, unused windows have been closed. Even though we do not have evidence on how much more efficient developers are when they are confronted with less windows, the automatic closing of windows provided by *AutumnLeaves* certainly helps developers to more quickly gain an overview of their workspace and of the subject system and to hence ease the source space navigation and exploration.

	Session 1	Session 2	Session 3	Session 4	Session 5	Average
Number of opened windows	82	41	109	33	61	65.20
Correctness (with some later window usage permitted)	74.18%	51.26%	47.52%	59.74%	80.20%	61.61%
Correctness strict	53.33%	40.00%	46.29%	52.94%	63.63%	51.76%
Number of windows incorrectly closed (false-positives)	7	15	29	8	12	13.50
Number of windows incorrectly not closed (false-negatives)	8	4	11	3	7	6.12
Time elapsed between last usage and closing [minutes:seconds]	8:12	7:52	12:56	9:06	5:34	10:09
Avg. number of windows without *AutumnLeaves*	25.20	15.86	32.50	19.94	27.34	28.53
Avg. number of windows with active *AutumnLeaves*	17.84	8.41	18.88	10.20	11.95	26.03
Delta in avg. number of windows *AutumnLeaves*	7.36	7.45	13.62	9.74	15.39	12.50

Table IV
CORRECTNESS, FALSE-POSITIVES, FALSE-NEGATIVES AND AVERAGE NUMBER OF WINDOWS IMPROVEMENTS PROVIDED BY *AutumnLeaves* OF FIVE RANDOMLY SELECTED SESSIONS AND AVERAGED OVER ALL 25 SESSIONS.

Another interesting result would certainly be the navigation time, that is, whether less windows indeed reduce the navigation time. We have not yet evaluated enough data to obtain significant results, but early evaluations indicate that the navigation time and effort is lower with less windows open. In future work we address this question in more detail.

Task-dependent results. The task-dependent evaluation we performed revealed that both correctness and effectiveness (window reduction) depend on the nature of the task. We obtained the highest correctness values for new feature implementation and defect correction tasks (non-strict correctness of 67.37% averaged over all such tasks). However, for these tasks the reductions of windows was, at 9.46 windows, below the average of all 25 sessions (12.5 windows). For tasks concerned with implementing a new system, performance optimizations or pure navigation, the correctness was lower (59.86%) and the reduction rate higher (13.85 windows). We attribute these results to the fact that tasks in which developers mostly navigate a constrained part of the system require opening fewer windows than tasks involving navigation of several, possibly unrelated parts of the system. *AutumnLeaves* can more correctly but less often identify obsolete windows when the general focus is on statically strongly related entities. Furthermore, feature implementation and defect correction tasks encompass heavily the use of structural relationships between source artifacts (*e.g.*, inheritance), thus *AutumnLeaves* can more correctly identify related windows. We leave as future work to find means for weight propagation based on non-structural information to obtain better performance for the other kind of tasks such as exploration tasks.

Variations. We tested the effect of weight propagation and different threshold mechanisms with two different experiments: i) not considering propagation of weight and ii) using two threshold instead of just one.

In the first experiment we ran two benchmarks: one using all weights as determined in the first experiment, including propagation, and another one omitting propagation of weight. The latter experiment gives slightly lower values for both, correctness and reduction of average number of windows (2.56% less correct and 5.46% less reduction of number of windows). Thus we consider propagation of weight to related entities as important, although its effect is not huge.

Instead of rewarding on each user action windows that are fully or partially visible, we evaluated in the second experiment a variation of *AutumnLeaves* which defines two thresholds, a lower boundary for visible windows and a higher boundary for hidden windows. As the latter are more likely to not be useful anymore, we assume that they can vanish earlier. For this experiment we have chosen a lower threshold of 20% of the average window weight and an upper threshold of 40% (compared to the standard threshold of 30%). The results of this experiment averaged over all 25 session are the following: correctness slightly increased to 63.58% while strict correctness (no later window usage permitted) dropped to 50.94%. The delta of average number of windows increased to 14.3 windows, while false-negatives and false-positives did not show significant changes. We conclude that this variation did not yield remarkably better results.

Threats to validity. There are several threats to validity in the experiment we performed. Firstly, the data sets we used cover pretty short development sessions (up to three hours) and were concerned with rather simple and constrained tasks. Large industrial projects may encompass longer and more complex and open tasks (threat to external validity). However, we consider these development activity logs as being fairly typically for medium-sized applications, in particular as there were four different applications involved. We can also assume that even if the tasks have been rather small and short in our data sets, the performance of *AutumnLeaves* nonetheless scales up to larger tasks as those are likely to have similar constraints and characteristics with respect to window usage.

Secondly, the fact that a window is not explicitly used anymore in the recorded data set is not necessarily a sign that it was not important later on (threat to construct validity). The developer could have looked at the content of such a window without interacting with it. At least in Squeak it is possible to have a window in the front and reading its content without ever selecting and giving it the focus. However, this

is not possible in Eclipse. Although such situations might have occurred in the recorded development sessions, we assume those to be very rare. Thus they should not have a significant influence on the reported results and on the prediction quality of *AutumnLeaves*. In both environments developers might have glanced at a window just shortly to find out that it the wrong one

Thirdly, developers that gave us the data sets also reported on the task they performed therein. From their description, we assigned each task to the six different task categories. We did not manually study the data sets or asked developers further, whether they worked on just one single task without any perturbations or whether they performed some other sub-tasks or unrelated work in this recorded activity log. Some descriptions of tasks were ambiguous as developers performed work not unmistakably assignable to one single task (threat to internal validity). Thus the task-dependent evaluation of *AutumnLeaves* contains some pitfalls regarding accuracy of the results, as the different performance of *AutumnLeaves* in some tasks is partially also explained with perturbations in the data sets and difficulties in assigning these sets to particular tasks. We consider this effect as marginal though. Another threat concerning task-dependent evaluation is that we did not have an equal distribution of the data sets on the six tasks. For instance, there was only one single data set concerned with performance optimization but seven data sets contained a navigation task. This imposes a serious threat to construct validity.

B. Practicality

While the results of the benchmark validation elaborate on the correctness of *AutumnLeaves*, the practical usefulness of our proposal is not assessed by such validation. We thus study the practicality of *AutumnLeaves* in this section.

From the discussion with developers, we learned that a crowded workspace with tons of open windows seriously hampers development efficiency, no matter on which task they are working. In particular when navigating software systems to reverse-engineer them, for instance to build a mental model of a system in order to be able to extend or correct particular software features, developers suffer from too many open windows, which can ultimately lead to a lost of overview. Any solution to overcome this window plague comes as a relief, developers reported. However, it is considered as important to have full control over the windows. Developers are not willing to accept a fully automatic closing mechanism, instead they always want to have the power of veto, for instance if *AutumnLeaves* suggests to close a window actually being used as an important reference point.

In developer interviews we also revealed an interest in visual clues about how important *AutumnLeaves* considers a window. This not only supports developers in estimating when a particular window will be removed, but is also helpful in locating windows still being actively used. *AutumnLeaves* currently visualizes the internally maintained weighting of windows by showing in the window title bar different colors. For windows considered as active (their weight is above the threshold), the title bar is colored in a heat gradient from red to blue, while red means very important, blue less important, as suggested by other researchers [6], [10]. Windows identified for closing are grayed out in a gradient from light gray to black, where black indicates a weight way below the threshold. Such visual clues also serve as a navigation aid to developers, as they often find the window of interest by looking at the title bar colors. In most cases an interesting window has a non-gray color and often even a red color.

As future work we leave to empirically determine the impact of different weights (as shown in II and Table III) in practice, the gain on productivity of *AutumnLeaves* or the correlation between the window importance computed by *AutumnLeaves* and what developers themselves consider as important windows.

V. Differences between IDEs

As the two IDEs, Eclipse and Smalltalk, have fundamental differences in their window management (as mentioned in Section II), we expect differences as well regarding *AutumnLeaves*. Even though the Eclipse data sets do not significantly differ from the Squeak data sets, the low number of Eclipse data sets (3 compared to 22 for Squeak) does not allow us to draw a statistically relevant conclusion yet. Generally, the *AutumnLeaves* algorithms are less complex in Eclipse as for instance visibility is just a boolean variable — either a tab is visible in the tab bar or it appears in the tab list (making it essentially invisible). Moving and resizing of windows is not relevant in Eclipse (compared to Squeak).

Usage data shows, however, that in Eclipse more windows are open on average (see Section II), probably due to the fact that Eclipse only supports limited navigation in windows (for instance, we cannot open a new class in an existing window). So far we have not analyzed enough data sets from Eclipse to judge whether we have to adapt considerably the *AutumnLeaves* algorithms or the weighting mechanism to adapt to the navigation differences in Eclipse compared to Squeak. The data we analyzed gives us the impression that *AutumnLeaves* is robust enough to also properly handle Eclipse window management. Further work aims at gathering and analyzing more Eclipse development data. At the moment we are optimistic that *AutumnLeaves* requires only fine-tuning of, for instance, the weighting procedure to work equally well in Eclipse as in Squeak.

VI. Related Work

Development environments, particularly Seesoft [6], FEAT [20], NavTracks [21] and Mylyn [10], aim at a goal similar to that of our proposal, that is, improving

the navigation of the source space and thus easing reverse engineering and understanding software systems. However, there are several fundamental differences to our proposal.

The Seesoft software visualization system [6] eases software analysis by mapping each line of code to a colored row. The color indicates an interest metric: red lines are for instance most recently changed lines and blue lines least recently changed. Seesoft is not able to reason about navigation activities in the IDE. *AutumnLeaves* works at a higher level than source code lines. We analyze window and entity usage in development sessions to reduce the number of windows and to thus help developers to avoid being distracted from their development focus by obsolete windows.

FEAT [20] applies a concern graph to visualize scattered but conceptually related code elements together in order to navigate concerns. However, in the original FEAT tool, developers had to manually create this concern graph. Robillard *et al.* [19] enhanced FEAT to automatically infer concerns, however, users still have to accept or decline the inferred concerns; our approach does not require any explicit user action, but does not reason about software concerns.

NavTracks [21] exploits the navigation history to recommend files related to the file the developer is currently looking at. This approach works at the granularity of files, hence does not take into account specific methods or classes. However, a recommendation list helps little to obtain an overview over the whole system; the developer just sees a list of artifacts possibly related to a specific artifact, but is not supported in locating all interesting entities in a "big picture" view. With this approach, developers may still be overloaded by tons of windows and thus may have difficulties to find their way in the workspace.

Mylyn (formerly known as Mylar) [10] computes a degree-of-interest value for each source artifact based on the historical selection or modification of the artifact. The background color of the artifacts highlights their relative degree-of-interest in the context of the current task — interesting entities are assigned a "hot" color. In Mylyn the information used to compute the interest value is relatively simple: selecting and editing an artifact increases the interest; if no further event occurs the interest decreases over time. In our approach, we propose to also take into account removal and creation of entities. Furthermore, we propagate weight, our measure for importance, to statically related entities. While Mylyn purely reasons about software artifacts, we put the focus on windows showing these entities, and visualize the importance of windows, not of source artifacts. By removing unimportant windows, we indirectly support developers to more quickly locate entities of interest in their reduced working set of windows. Similarly Strathcona which focuses on recommending examples using structural dependencies [8] does not propagate weights and uses heuristics to identify examples, it does not support the automatic closing of windows.

Fluid source code views are related to our approach in the sense that they allow programmers to fluidly shift attention to related source code entities [3]. As such fluid source code views reduce the need for programmers to navigate and to open extra windows. Still we believe that our approach is orthogonal since it focuses on the windows opening plague. Combining *AutumnLeaves* and fluid source code views is a definitively interesting future work.

Other researchers tackle the problem of software navigation by providing high-level visualizations of the system, either focusing on static system structure or dynamic system behavior, or both. Usually such visualizations are not directly integrated in development environment but accessible in separated tools. Löwe *et al.* [14] for instance merged information from static analysis with information from dynamic analysis to generate visualizations. Reiss [17] visualizes the dynamics of Java programs in real time, *e.g.*, the number of message sends received by a class. Program Explorer [12] provides interactive visualizations of design patterns to better navigate and understand frameworks. Moose [15] is a software analysis platform encompassing various software visualizations such as different polymetric views [13] supporting reverse engineering tasks.

VII. Conclusions and Future Work

In this paper we studied the window plague occurring in most modern IDEs such as Eclipse or Squeak Smalltalk. We analyzed several development sessions of various developers to reveal the extent and graveness of workspaces crowded with plenty of windows. Developers remarked that an automatic means to close windows is beneficial for them and thus we built *AutumnLeaves*, a mechanism that observes all open windows and how they are related to each other by associating a weight to each window. This weight reflects the current importance of a window and its content (classes or methods) and thus *AutumnLeaves* can identify obsolete windows that are most likely not useful anymore in the current development session. *AutumnLeaves* hence automatically closes this window, if developers do not decline this. We evaluated *AutumnLeaves* with a benchmark validation analyzing 25 recorded development sessions to determine the correctness of *AutumnLeaves*' algorithms. The correctness results reveal that *AutumnLeaves* is usually able to pinpoint the windows that are appropriate candidates for closing. We further reported on the practicability of our approach and critically discussed it.

In the future we aim at extending *AutumnLeaves* to cover all kinds of windows opened in an IDE, also windows containing non–source files such as configuration files or XML documents. For this purpose we have to find means to relate such content to traditional source artifacts, for instance by parsing the content to locate names of classes or methods so that we find references to already open windows. Another

goal for the future is to experiment with other weights for the various actions or with other mechanisms to associate these weights, for instance to assign weight based on the extent of modification occurring in an entity and a window, or to steadily evaporate weights of windows not being used for a longer period of time, for instance to close them more quickly. Furthermore, we plan to gather more empirical evidence from developers, to for example answer questions such as whether people agree with the closing suggestions drawn by *AutumnLeaves* or whether human beings can themselves faster or more correctly identify obsolete windows. This also includes performing a large controlled experiment to evaluate the impact of *AutumnLeaves* on productivity in practice.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010) and of ESUG (the European Smalltalk User Group) http://www.esug.org/

REFERENCES

[1] V. Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1):3–12, 1997.

[2] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[3] M. Desmond, M.-A. Storey, and C. Exton. Fluid source code views. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 260–263, Washington, DC, USA, 2006. IEEE Computer Society.

[4] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[5] Eclipse platform: Technical overview, 2003. http://www.-eclipse.org/whitepapers/eclipse-overview.pdf.

[6] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992. Depth.

[7] A. Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of ICSE'05*, pages 1–10, 2005.

[9] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.

[10] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[11] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 125–135, 2005.

[12] D. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 342–357, New York NY, 1995. ACM Press.

[13] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[14] W. Löwe, A. Ludwig, and A. Schwind. Understanding software - static and dynamic aspects. In *17th International Conference on Advanced Science and Technology*, pages 52–57, 2001.

[15] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[16] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, volume 0, pages 13–22, Los Alamitos CA, 2006. IEEE Computer Society.

[17] S. P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.

[18] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*, pages 317–326, 2008.

[19] M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–234, Oct. 2003.

[20] M. P. Robillard and G. C. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of 25th International Conference on Software Engineering*, pages 822–823, May 2003.

[21] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, Washington, DC, USA, sep 2005. IEEE Computer Society.

[22] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.